

# **SANDIA REPORT**

SAND2001-8338

Unlimited Release

Printed June 2001

## **The Generalized Security Framework**

Richard J. Detry, Stephen D. Kleban, Patrick C. Moore

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of  
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from  
National Technical Information Service  
U.S. Department of Commerce  
5285 Port Royal Rd  
Springfield, VA 22161

NTIS price codes  
Printed copy: A03  
Microfiche copy: A01



SAND2001-8338  
Unlimited Release  
Printed June, 2001

## The Generalized Security Framework

Richard J. Detry, Stephen D. Kleban, Patrick C. Moore

Sandia National Laboratories  
Albuquerque, NM 87185

### **ABSTRACT**

The Generalized Security Framework (GSF) consists of a set of libraries, classes, and tools that provide developers with the ability to easily secure distributed applications and collaborative environments. The GSF uses and enhances the Generic Security Services API (GSSAPI) to provide authentication, authorization, data protection, delegation, and auditing. It currently works with either DCE or Kerberos as the underlying security mechanism, and it has been designed so support for PKI can be easily added in the future. DCE/Kerberos is a scaleable, mature, robust security infrastructure embraced and accredited throughout the Nuclear Weapons Complex (NWC) for a secure collaborative modeling and simulation environment. The goal of the GSF is to provide a common security foundation that can be applied and extended to create secure distributed applications, independent of the communications protocol.

The GSF provides a number of extensions that embed GSF security in specific remote communication APIs, such as Java sockets and Java RMI. The extensions have been designed and implemented in such a manner as to require minimum changes to application code in order to move from an unsecure application to a secure application. The advantage of this approach is that security can be enforced reliably and consistently since very little is required of the application developer. In this paper, the authors describe the goals, design, and implementation of the Generalized Security Framework.

This page intentionally left blank

# Table of Contents

<b>1</b>	<b>INTRODUCTION .....</b>	<b>7</b>
<b>2</b>	<b>DCE/KERBEROS INFRASTRUCTURE .....</b>	<b>8</b>
<b>3</b>	<b>GENERIC SECURITY SERVICES API .....</b>	<b>9</b>
3.1	ESTABLISH GLOBAL IDENTITIES .....	10
3.2	ESTABLISH A SHARED SECURITY CONTEXT .....	10
3.3	TRANSFER DATA .....	10
3.4	DESTROY THE SECURITY CONTEXT .....	11
3.5	DELEGATION .....	11
3.6	SHORTCOMINGS OF THE GSSAPI .....	12
<b>4</b>	<b>ADDITIONAL GSF FEATURES .....</b>	<b>12</b>
4.1	OBTAINING INFORMATION ABOUT THE INITIATOR .....	12
4.2	AUTHORIZATION .....	13
4.3	AUDITING .....	13
4.4	ESTABLISHING A DELEGATED CREDENTIAL .....	13
4.5	OBTAINING VALUES OF REGISTRY ATTRIBUTES .....	13
<b>5</b>	<b>GSF DESIGN .....</b>	<b>14</b>
5.1	C++ LAYER .....	14
5.2	JAVA LAYER .....	15
<b>6</b>	<b>GSF EXTENSIONS .....</b>	<b>16</b>
6.1	SECURING SOCKET APPLICATIONS .....	16
6.1.1	<i>GsfServerSocket</i> .....	16
6.1.2	<i>GsfSocket</i> .....	16
6.1.3	<i>GsfOutputStream</i> .....	17
6.1.4	<i>GsfInputStream</i> .....	17
6.1.5	<i>Using the Socket Extension</i> .....	18
6.2	SECURING JAVA RMI APPLICATIONS .....	18
6.3	SECURING CORBA APPLICATIONS .....	18
<b>7</b>	<b>GSF WEB .....</b>	<b>20</b>
<b>8</b>	<b>CUSTOMERS .....</b>	<b>21</b>
8.1	DISTRIBUTED RESOURCE MANAGEMENT .....	21
8.2	SAMPLLL .....	21
8.3	COMPASS .....	21
<b>9</b>	<b>FUTURE WORK .....</b>	<b>21</b>
<b>10</b>	<b>CONCLUSION .....</b>	<b>22</b>
<b>11</b>	<b>REFERENCES .....</b>	<b>22</b>

## **ACRONYMS**

API – Application Programming Interface  
ASCI – Accelerated Strategic Computing Initiative  
CoMPASS – Confederation of Models to Perform Assessments of Stockpile Stewardship  
CORBA – Common Object Request Broker Architecture  
DCE – Distributed Computing Environment  
DFS – Distributed File System  
DOE – Department of Energy  
DRM – Distributed Resource Management  
GSF – Generalized Security Framework  
GSSAPI – Generic Security Service Application Programming Interface  
IDL – Interface Definition Language  
IETF – Internet Engineering Task Force  
JSP – Java Server Pages  
MIT – Massachusetts Institute of Technology  
NWC – Nuclear Weapons Complex  
PKI – Public Key Infrastructure  
RMI – Remote Method Invocation  
SAMPL – Simplified Analytical Model of Penetration with Lateral Loading

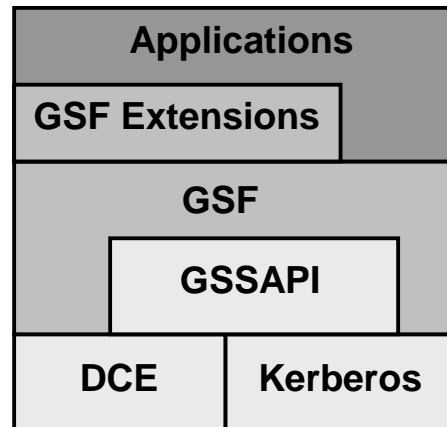
# 1 Introduction

At Sandia National Laboratories and collaborating DOE facilities, distributed applications are being developed in order to provide desktop access to modeling and simulation applications that make use of shared, high-performance compute and data resources. Most of these distributed applications are being developed in object oriented languages, such as Java and C++, using modern protocols and middleware such as Java sockets, Java RMI, and CORBA. But the compute resources and the information they serve are required by DOE regulations and laboratory policies to be strictly controlled. Therefore, to be useful, these applications must be highly secured, typically to a level commensurate with operation on classified networks within the DOE Complex. The purpose of the Generalized Security Framework (GSF) is to enable distributed applications operating in a heterogeneous environment to be secured simply, reliably, and consistently.

The GSF consists of a set of libraries, classes, and tools that provide developers with the ability to easily secure distributed applications and collaborative environments developed with either C++ or Java. The GSF uses and enhances the Generic Security Service Application Program Interface (GSSAPI) to provide authentication, authorization, data protection, delegation, and auditing. It currently works with either DCE or Kerberos as the underlying security mechanism, and it has been designed so support for PKI or other security mechanisms can be easily added in the future. DCE/Kerberos is a scaleable, mature, robust security infrastructure embraced and accredited throughout the DOE Complex for a secure collaborative modeling and simulation environment. The goal of the GSF is to provide a common security foundation that can be applied and extended to create secure distributed applications, independent of the communications protocol.

The GSF also provides a number of extensions that embed GSF security in specific remote communication APIs, such as Java sockets and Java RMI. These technologies are familiar and attractive to developers, who unfortunately are often unaware that they are fraught with security risks: the network connections are not authenticated and data is passed over these connections as clear text. Occasionally the developer has invested significant design and development effort before learning that the application is unacceptable from a security perspective. The GSF extensions have been designed in such a manner as to require minimum changes to application code in order to provide authentication, data integrity, and data privacy to these remote communications. The advantage of this approach is that security can be enforced reliably and consistently since very little is required of the application developer. Additionally, we can frequently ‘rescue’ an application that has been developed with insufficient attention to security.

Figure 1 provides a high-level diagram of the GSF. The remainder of this paper will describe each of these components in more detail.



**Figure 1 A high-level diagram of the Generalized Security Framework**

## **2 DCE/Kerberos Infrastructure**

The DOE laboratories, in conjunction with DOE accreditation authorities, have invested years in the selection, testing, and accreditation of a security infrastructure. This infrastructure is grounded on the use of Kerberos [1,2] as an authentication mechanism. In addition, the labs have deployed and cross-certified their Kerberos infrastructure, which is based upon the Distributed Computing Environment (DCE) [3]. In addition to Kerberos authentication services, which are based upon MIT Kerberos Version 5 [4], DCE provides authorization services, directory services, and a secure Distributed File System (DFS). DCE cells have been implemented at numerous sites and are accredited by DOE for cross-cell authentication over SecureNet (the network that connects classified environments at the DOE research laboratories and manufacturing facilities). Thus a user may log in to their home cell, acquire DCE credentials, and use those credentials to log in to resources and access data at the other sites without an additional log in. DCE authorization services provide access controls to DCE objects and DFS files based on a standard model. Furthermore, these access controls may be fully distributed, so that users and groups at one site may be allowed to control access to objects or files that are hosted at another. This provides a convenient and appropriate security model for inter-site access.

One of the advantages of DCE is that, being based upon Kerberos, it can also serve as a Kerberos Key Distribution Center. Since Kerberos source is available and it has been ported to a larger number of platforms than DCE, including Linux and Macintosh, it is possible for these additional platforms to use the DCE security infrastructure via the Kerberos libraries and obtain an important subset of the DCE functionality. This also allows applications to be secured using pure Kerberos or DCE without having multiple security infrastructures.

### 3 Generic Security Services API

The GSF accesses the underlying security mechanism primarily through the Generic Security Service Application Program Interface (GSSAPI). The choice of GSSAPI as the underlying abstraction for GSF naturally followed from the fact that we have a DCE/Kerberos security infrastructure and both DCE and Kerberos provide GSSAPI interfaces. Irrespective of that fact, we feel that GSSAPI has proven to be an excellent basis for the Generalized Security Framework because:

- GSSAPI is based on an Internet Engineering Task Force (IETF) standard [5,6] and is the basis for a number of IETF common authentication technologies.
- GSSAPI is available in many commercially supported security products (DCE, Solaris, Entrust, CyberTrust). Microsoft Windows 2000 provides a GSSAPI-like API, and at the network token level, it is GSSAPI compliant [7].
- GSSAPI is freely available in source distributions, including Globus/GSI [8,9] and MIT Kerberos[4].
- GSSAPI makes it possible to port an application from one infrastructure/mechanism to another. In at least one case (Globus) it is possible to convert a distributed application from a PKI based infrastructure to a Kerberos based infrastructure simply by relinking the application.
- GSSAPI provides documented and standard security abstractions that can be understood by developers, and once understood will be applicable to most security infrastructures and environments.

The basic security abstractions provided by GSSAPI are:

- Principal or global identity - something or someone who can be authenticated as a participant in a secure conversation.
- Credential – something that can be acquired by an object that wishes to assume a global identity.
- Security context – a state shared by two ends of a secure conversation.
- Initiator – an object that initiates a security context.
- Acceptor – an object that accepts a security context.
- Delegated credential – something that can optionally be forwarded by an initiator in order for the acceptor to act on behalf of the initiating identity.
- Confidentiality – a property that causes the conversation between an initiator and acceptor to be encrypted to prevent eavesdropping of data in transit.
- Integrity – a property that causes the conversation between an initiator and acceptor to be secure against tampering with data in transit.
- Mutual Authentication – a property that assures that if the initiator specifies the identity of the acceptor, and the acceptor specifies the identity of the initiator, no conversation will be established unless both are securely authenticated.

There are four phases involved in a typical GSSAPI application:

- establish global identities,
- establish a shared security context,
- transfer data
- destroy the security context.

### **3.1 Establish Global Identities**

Before attempting to establish a context, both the initiator and the acceptor must establish their global identities and obtain credentials. A human user accomplishes this step by performing either a `dce_login` or a `kinit` from the keyboard. A server obtains its credentials from a key table (usually called a `keytab`) file. This step is accomplished programmatically from within the GSSAPI.

### **3.2 Establish a Shared Security Context**

Once the initiator and acceptor have obtained credentials, they can establish a shared security context. The initiator begins the process by making a GSSAPI call to initialize a context. Parameters to this call include the principal name of the acceptor with which the context is to be established and the requested options for the context. The most common options include mutual authentication and delegation. The GSSAPI routine returns a cryptographically protected, opaque token that contains information that is used by the acceptor to identify and authenticate the initiator. This token must be sent to the acceptor using the transport protocol or middleware that is being used by the application.

The acceptor takes the client's authentication token and passes it into another GSSAPI routine to accept the context. This routine essentially authenticates the initiator. It returns an opaque token to the acceptor that can subsequently be used by the initiator to authenticate the acceptor. This is only required if the initiator has requested mutual authentication. This token must therefore be sent back to the initiator, which then makes another call to the GSSAPI to authenticate the acceptor. In some cases, the process of obtaining and exchanging authentication tokens must be repeated until enough information has been obtained for each side to authenticate the other.

Once each party has been authenticated, each side has a context id that identifies the established security context. This context id can then be used to protect the data that is to be transmitted between the initiator and the acceptor.

### **3.3 Transfer Data**

Once a secure context has been established between the initiator and the acceptor, they can transfer data securely. The GSSAPI per-message security services can provide either of the following:

- integrity and authentication of data origin
- confidentiality, integrity, and authentication of data origin

In the first case, a cryptographical signature of the data is generated and passed along with the data to the peer. The peer then verifies that the signature matches the data, which ensures that the data was signed using the shared security context and that the data has not been modified, intentionally or otherwise, during transit. In the second case, the original data is cryptographically sealed (encrypted) and then passed to the peer, who subsequently unseals (decrypts) the data. If the data is successfully unsealed, the application can be sure that the original data was sealed using the shared security context and that the data has not been modified or viewed while in transit. The process of sending and receiving protected messages between the initiator and acceptor can continue as long as the shared security context remains valid. The context will remain valid until it expires or is destroyed.

### **3.4 *Destroy the Security Context***

Once the initiator and acceptor have finished exchanging data, the context should be destroyed in order to free system resources and ensure that the context can no longer be used. The most common way to destroy a context is to have the initiator make a GSSAPI call that destroys the context. This call returns an opaque token that is passed to the acceptor, which passes the token into another GSSAPI routine that destroys the context on the acceptor side. Another approach to destroying the context is simply to have both the initiator and the acceptor directly make the GSSAPI call to destroy the context. This approach does not require the transmission of a token. At this point, either approach can be used. In the future, however, the GSSAPI might require that a token be passed between the initiator and acceptor to properly destroy the context.

It is important to note that all information regarding the context resides in the application's process memory. Therefore, when the application stops running, gracefully or otherwise, the contexts that it has established are no longer valid.

### **3.5 *Delegation***

Another important concept, especially in distributed systems, is delegation. The initiator can delegate rights to allow the acceptor to act as its agent. Delegation means the initiator gives the context acceptor the ability to initiate additional security contexts as an agent of the initiator. To delegate, the context initiator sets a flag on the call to the GSSAPI routine to request a context, indicating that it wants to delegate. The initiator then sends the returned token in the normal way to the acceptor. When the acceptor passes this token to the GSSAPI routine to accept the context, a delegated credential handle is generated. The acceptor can then use this credential to initiate additional security contexts.

### **3.6 Shortcomings of the GSSAPI**

The GSSAPI is very useful and provides an excellent foundation for securing distributed systems, but it does have a few shortcomings:

- The GSSAPI provides no authorization or auditing capabilities. For many systems, enforcing need-to-know requirements is essential, and having audit trails available is highly desirable, sometimes critical. There are works-in-progress, including IETF drafts for standardizing a “Generic Authorization and Access control Application Program Interface “(GAA-API)
- It does not take full advantage of the underlying security mechanism. Since the GSSAPI was explicitly designed to be generic and applicable to many security mechanisms, mechanism-specific features are not accessible via the GSSAPI. This might not be an issue for many applications, but others frequently require access to these features.
- The implementations of the GSSAPI are inconsistent. Some features implemented by one security mechanism are not implemented by others, and the different security mechanisms frequently use different data types.
- The current GSSAPI offers an incomplete interface for exporting a security context with a delegated credential. This will hopefully be resolved in future enhancements to the standard.
- It is not object oriented and provides no direct support for Java. Most modern distributed applications use object oriented languages and middleware, which makes it difficult to use the GSSAPI. This is especially true of applications written in Java, which can only access the GSSAPI implementations via a Java Native Interface (JNI) layer [10,11].

The GSF addresses these shortcomings by providing greater access to the underlying security mechanism, adding a logging capability, hiding the differences between the GSSAPI implementations, and bridging the gap between the GSSAPI and object oriented languages and middleware.

## **4 Additional GSF Features**

The GSF provides many features above and beyond those available directly through the GSSAPI. The availability of features is, of course, dependent upon the underlying security mechanism. For example, DCE provides many features above and beyond those available when using Kerberos, whereas Kerberos provides superior performance. The GSF makes it very simple to switch between underlying security mechanisms, but specific security or performance requirements typically dictate which security mechanism should be used. This section describes the additional security features provided by the GSF and to which security mechanism the features are applicable.

### **4.1 Obtaining Information about the Initiator**

The GSF provides a simple way for the acceptor to obtain information about the initiator, including his principal name, local name, and realm. If DCE is being used as the underlying security mechanism, the groups to which the initiator belongs can also be obtained. The group information can be used to make simple access control decisions.

## **4.2 Authorization**

When DCE is used as the underlying security mechanism, the GSF provides a robust authorization utility that can be used to enforce need-to-know requirements. Access control lists (ACLs) are stored in the DCE Cell Directory Service (CDS) as directories and objects. Typically, a CDS directory corresponds to a server, while a CDS object below the server directory corresponds to a server method. In this fashion, per-server or per-method access control can be provided. This approach provides great flexibility in implementing access control for a distributed application and allows the resource owner to explicitly allow or deny any local or foreign user, group, or delegate.

## **4.3 Auditing**

Regardless of which security mechanism is being used, the GSF provides a simple logging capability. A time-stamped message is written to a log file for each security relevant event, including

- Context establishment
- Results of an authorization check
- Context deletion
- Delegation

These log files can be audited as necessary to look for discrepancies or simply to determine usage. Applications can also write messages to the log file as necessary.

## **4.4 Establishing a delegated credential**

Delegated credentials are a GSSAPI opaque abstraction that must be protected and handled in a mechanism-specific manner when established for use by an acceptor that wishes to act as a delegate on the client's behalf. In the GSF, applications have methods available for creating a login context (DCE) and exporting a credential (Kerberos) to support this. These methods establish a login context so that the acceptor may access DCE objects or execute applications as if the user or user-delegate was currently logged in.

Establishing a delegated credential must be done carefully in multithreaded environments because an established login context is process-wide, not thread-specific. Mutexes, synchronized java methods, and forked sub-processes are used to assure that execution threads never have unintended access to another thread's established context.

## **4.5 Obtaining Values of Registry Attributes**

Another option that is available when using DCE is the ability for the acceptor to obtain the value of a registry attribute for the initiator. A registry attribute is an attribute that is associated with a principal's account. These attributes are defined, created and set by DCE administrators, then the value of the attribute can be set for each principal in the cell. One such attribute used at Sandia defines the principal's DFS home directory, which can be used by a server to securely access or store information on behalf of the principal.

## 5 GSF Design

Since the GSF primarily uses the GSSAPI to access the functionality of the underlying security mechanism, it should come as no surprise that the GSF class names frequently include common GSSAPI terms such as initiator, acceptor, and context. The main GSF classes and their purpose are summarized below:

- *GsfContext* - contains the methods that are common to both acceptors and initiators, such as the data protection methods and the methods to destroy the context.
- *GsfInitiator* - contains the methods that are used by a context initiator to establish secure contexts with an acceptor
- *GsfAcceptor* - contains the methods used by a context acceptor to initialize itself, accept contexts, obtain info about the initiator, and authorize the initiator.
- *GsfBuffer* - used to pass data to and retrieve data from a number of other GSF methods; simplifies the interaction with the opaque tokens used by the GSSAPI.
- *GsfInitiatorData* - provides information to an acceptor about the initiator, including global name, local name, realm, and group membership.
- *GsfException* - the fundamental GSF exception. Most exceptions thrown by the GSF are of this type, although other exceptions are defined and thrown in cases where the application might want to identify and respond to a particular situation. The GSF frequently captures GSSAPI errors and translates them so they have some meaning to an application developer or user.

The class names are the same in both the C++ and Java version of the GSF. Using these classes, security can be easily added to an application independent of protocol.

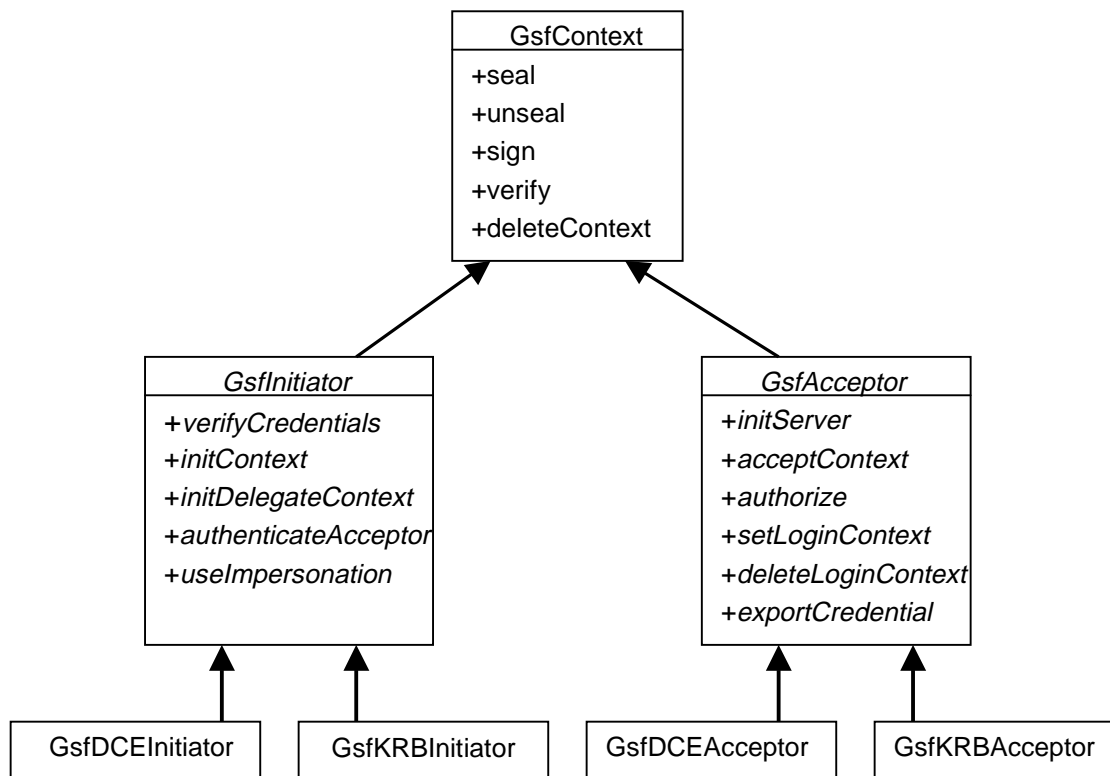
### 5.1 C++ Layer

The bulk of the GSF code is written in C++ in order to interact with the DCE and Kerberos libraries, which are written in C. The *GsfInitiator* and *GsfAcceptor* classes are abstract: they simply define the interface used by the application. The implementation of the interfaces is contained in classes extended from *GsfInitiator* and *GsfAcceptor*, as shown in Figure 2.

The *GsfContext* class is concrete. The implementation of its methods is not dependent upon the underlying security mechanism. The *GsfInitiator* and *GsfAcceptor* classes are abstract because the implementations of these methods are typically dependent upon the underlying security mechanism. For example, the DCE GSSAPI contains functions that are not part of the GSSAPI standard. One such function must be called in order to obtain a handle to the user's credentials. Another function provides a bridge between the GSSAPI and the DCE API, which is used to access additional features provided by DCE.

The appropriate initiator and acceptor classes are created by a corresponding singleton factory class. There are generic *GsfInitiatorFactory* and *GsfAcceptorFactory* classes that contain *create* and *destroy* methods. These methods are implemented in the appropriate child classes:

*GsfKRBInitiatorFactory*, *GsfDCEInitiatorFactory*, *GsfKRBAcceptorFactory*, and *GsfDCEAcceptorFactory*.



**Figure 2 Simplified class diagram for the GSF**

This design makes it simple to add support for additional security mechanisms should the need arise. A few simple factory classes would have to be created and implementations of the *GsfInitiator* and *GsfAcceptor* interfaces would have to be provided. Adding support for an additional security mechanism would remain hidden from the application developers. They could switch to a different underlying security mechanism by simply compiling with a new GSF library.

Additional classes in the C++ layer provide support for exception handling, data handling, logging, debugging, and authorization.

## 5.2 Java Layer

The Java version of the GSF uses the C++ version via the Java Native Interface (JNI). Therefore, the GSF Java code is generally a thin layer that simply dispatches the request to the C++ library via the JNI. In addition, the Java layer performs various sanity checks as well as translation of exceptions thrown in the C++ code.

The JNI is a powerful, flexible Java feature that allows developers to take advantage of the Java platform, but still utilize code written in other languages. Using the JNI, however, can be tricky and requires a bit of practice and experimentation. One of the greatest features of the GSF is that it provides this JNI layer so application developers using Java can easily take advantage of the powerful security features provided by DCE and Kerberos.

## 6 GSF Extensions

The GSF classes described above can be used to secure a distributed application, independent of the underlying middleware or protocol. However, since most of our target applications are being developed using sockets, Java RMI [12], or CORBA [13], we have also developed extensions that enable the GSF to be used with these technologies in a mostly transparent manner. The application developers can create their application without worrying about security. Then, when the application is stable, a few simple code changes can be made to incorporate the GSF into the application. The developers are happy because they don't have to worry about security, and the security folks are happy because they have a high degree of confidence that the application has been secured correctly due to their familiarity with the GSF.

### 6.1 Securing Socket Applications

Due to its simplicity, the use of Java sockets seems to be on the rise. Due to the object-oriented nature of Java, we were able to develop an extension that incorporates the GSF into socket-based applications with only a few changes to the application code. The main classes that comprise the extension are discussed below.

#### 6.1.1 *GsfServerSocket*

This class extends *java.net.ServerSocket* and provides an implementation of the *accept* method, along with a new constructor. The constructor adds a number of parameters to those expected by the *ServerSocket* class. These additional parameters provide the necessary information for the server to be initialized so that it can accept secure connections from clients. The implementation of the *accept* method calls the *ServerSocket.accept* method, but rather than returning the resulting *java.net.Socket*, it creates and returns a *GsfSocket* object. This object is subsequently used by the application to obtain input and output streams that are used to securely exchange data.

#### 6.1.2 *GsfSocket*

The *GsfSocket* class is a wrapper class for *java.net.Socket*. It contains a number of constructors for creating *GsfSocket* objects for use on both the client and server sides. It also provides implementations of the *getOutputStream*, *getInputStream*, and *close* methods.

The constructors for creating a *GsfSocket* on the client side work in close cooperation with the constructors for creating a *GsfSocket* on the server side in order to exchange the tokens necessary for mutual authentication and to establish a shared security context. The sequence is initiated when the server calls the *GsfServerSocket.accept* method. The implementation of this method calls *java.net.ServerSocket.accept*, which blocks until data is written to the socket. This occurs

when the client constructs a new *GsfSocket* in order to communicate with the server. The implementation of the *GsfSocket* constructor calls the regular *java.net.Socket* constructor, which results in a regular Java socket being created between the client and the server. Once the regular Java socket is available, the implementation of *GsfServerSocket.accept* uses the socket to construct a new *GsfSocket*. At this point, the constructor for the *GsfSocket* on the client side and the constructor for the *GsfSocket* on the server side are able to use the regular Java socket to exchange tokens generated by calls to the GSF to establish a shared security context.

Once the shared security context has been established, the GSF-secured socket can be used to securely exchange data. The client and server each call the *getInputStream* and *getOutputStream* methods on the *GsfSocket* object. The implementation of these methods construct *GsfInputStream* and *GsfOutputStream* objects, respectively, and return references to these objects. These objects are then used by the client and server just as regular *java.io.InputStream* and *java.io.OutputStream* objects would be, but the data written to these streams is transparently protected.

When the client and server are done exchanging data, they each should call the *close* method on the *GsfSocket* object. The implementation of this method closes the regular Java socket and destroys the security context.

The *GsfSocket* class also contains a method that allows the server to obtain the *GsfAcceptor* object that is associated with the socket. The server can then call the methods available through the acceptor to perform authorization, export the user's credentials, etc.

### **6.1.3 *GsfOutputStream***

This class extends *java.io.OutputStream* and provides implementations of the three *write* methods. The *write* method that accepts a byte array, an offset and a length is called by the other two *write* methods. The implementation of this method determines the current protection level that should be applied to the data on this stream. The data can be sent in the open (no protection) or it can be signed or sealed. The protection level can be set via a call to the *setProtectionLevel* method, or it can be set via a parameter to the *GsfSocket* constructor. The implementation of the *GsfSocket.getOutputStream* method passes the desired protection level as a parameter to the *GsfOutputStream* constructor. The protection level can be changed at any time that the stream is open, which results in a very flexible output stream.

The *write* method begins by writing the protection level to the stream. It then applies the necessary level of protection to the data, if any, and then writes the data, the sealed data, or the data followed by the data's signature, to the stream.

### **6.1.4 *GsfInputStream***

This class extends *java.io.InputStream* and provides implementations of the three *read* methods. The *read* method that accepts a byte array, an offset, and a length is called by the other two *read* methods. The implementation of this method begins by reading an integer that identifies the level of protection that has been applied by the output stream to the available data. It then reads the data, then the signature of the data if necessary. If the data has been sealed, it is unsealed. If the data has been signed, a call is made to verify that the signature matches the data. Providing

that the *unseal* and *verify* methods are successful, the requested bytes of the original data are returned to the caller. If the *unseal* or *verify* methods fail, an *IOException* is thrown.

### **6.1.5 Using the Socket Extension**

Applications that have been developed using Java sockets can be secured using the GSF socket extension by including the GSF package (*gov.sandia.gsfx*) and changing two or three lines of code. On the server side, the call to the *java.net.ServerSocket* constructor is changed to be a call to a *GsfServerSocket* constructor. Similarly, on the client side, the call to the *java.net.Socket* constructor is changed to be a call to a *GsfSocket* constructor. If the application needs to change to level of protection applied to the data on the socket, an additional call can be made to change the protection level. That's all there is to it.

## **6.2 Securing Java RMI Applications**

Java RMI applications can be secured easily using a custom RMI socket factory [14] based upon the GSF socket extension discussed above. The GSF provides a *GsfRMIServerSocketFactory* class, which implements the *RMIServerSocketFactory* interface, and a *GsfRMIClientSocketFactory* class, which implements the *RMIClientSocketFactory* interface. The implementation of the application's remote interface (or interfaces) create instances of these factory classes and pass them to the constructor of their parent class, which will either be *java.rmi.UnicastRemoteObject* or *java.rmi.activation.Activatable*. The RMI runtime then calls the *createSocket* method of the client socket factory and the *createServerSocket* method on the server socket factory. The implementations of these methods create the necessary *GsfSocket* and *GsfServerSocket* classes, respectively. The RMI runtime then calls the *getInputStream* and *getOutputStream* methods on the *GsfSocket* classes, which results in the use of the *GsfInputStream* and *GsfOutputStream* classes. These streams then transparently apply the requested level of protection to the data. Server applications can also obtain the *GsfAcceptor* associated with the current client invocation in order to provide authorization, export credentials, etc.

Distributed applications can thus be developed using Java RMI without worrying about security. When the application is nearly complete, a few lines of code are changed and the application is secured via the GSF.

## **6.3 Securing CORBA Applications**

Due to the many CORBA products and the differences among them, we have not yet developed an extension to secure CORBA-based applications. We have, however, outlined a simple strategy that uses the GSF API directly to secure these applications. The strategy involves adding two methods to the server's IDL that can be invoked from the client side in order to establish a security context and destroy an established security context. The IDL for these methods and the necessary data types and structures is shown below.

```

typedef sequence<octet> RawData;

struct GsfAuthToken {
    long    acceptor_id;
    RawData signature;
};

RawData init_sec_context(in RawData token, out int index);

oneway void delete_sec_context(in GsfAuthToken token);

```

The RawData data type is used to exchange the opaque security tokens that are generated by the GSSAPI routines. The GsfAuthToken structure can be used to identify and authenticate a client in those situations where a server is simultaneously accepting invocations from multiple clients. The acceptor\_id is generally a key or an index into a data structure on the server side that is used to store the *GsfAcceptor* objects, each of which represents a security context with a client. The signature is generated on the client side by simply signing the acceptor\_id. This token is then sent to the server via an additional argument to every server method. The server obtains the token, looks up the associated GsfAcceptor using the acceptor\_id, and authenticates the client by verifying the signature. Note that if a server accepts connections with only one client, the use of the authentication token is not necessary.

In order to establish a shared security context, the client invokes the *initContext* method on a *GsfInitiator* object. This method returns an opaque token that is then sent to the server via the *init\_sec\_context* method. The server passes this token into the *acceptContext* method on a *GsfAcceptor* object, which authenticates the client and returns a token that is returned to the client. The client passes the token into the *authenticateAcceptor* method on the *GsfInitiator* object in order to authenticate the server. The server also returns an index, which is generally a key into a data structure that is used to store the *GsfAcceptor* objects. The client can sign the index and create a GsfAuthToken structure that will subsequently be sent to the server with every method invocation.

When the client is done invoking methods on the server, it destroys the security context by invoking the *delete\_sec\_context* method on the server. After this call, the security context is no longer valid.

Note that most of the above classes and calls can be hidden from the end developer via some stub and skeleton classes. Thus, for a large CORBA-based application, one person could develop these stubs and skeletons that could subsequently be used transparently by other developers. After a number of CORBA-based applications have been secured in the manner outlined above, we plan to look for patterns to see if we could provide some type of extension that further simplifies securing CORBA applications using the GSF.

## 7 GSF Web

There has been significant demand by developers for a system that allows Java servlets and Java Server Pages (JSP) to be secured using the GSF in order to provide secure access to distributed resources from a Web browser. This is not especially difficult in the special case where the servlet or JSP environment runs as a single network identity, typically an ‘entity’ account with the Kerberos key for that account stored in a keytab file that only the Java servlet environment can read.

But what developers really want is the ability for a servlet to impersonate the user running the web browser – i.e., to acquire that user’s DCE/Kerberos credential just as if they had done a kinit on the web server and run the servlet in their own user space. This is problematic for various reasons:

- Standard HTTP browsers do not support Kerberos authentication.
- Servlets and JSP run all user requests in a single multi-threaded environment.
- The Java runtime has a large footprint, making it inefficient to run multiple runtimes to support different users’ HTTP requests.
- DCE is multi-thread safe, but does not allow a single process to simultaneously impersonate more than one user.
- DCE threads may not be compatible with the threads package needed by the Java runtime.
- Kerberos is not entirely thread safe and was not designed to support multi-user sessions.

Nevertheless, we believe that within some constraints we can build a system that permits GSF-secured servlets and JSP’s that impersonate the end-user. We began piloting such a system in the spring of 2001. Our system uses HTTP basic authentication protected with SSL to provide Kerberos/DCE passwords to a modified Apache web server. This authentication method is standard within ASCII and the NWC. The Apache server converts the passwords to user credentials and connects to a Tomcat servlet engine running on the Web server host machine. The Tomcat engine has access, via a Java Native Interface (JNI), to a new, Web-specific initiator class called *GsfWebInitiator*. This class was designed such that a servlet always obtains the credential belonging to the authenticated user associated with the HTTP request.

We are working on ways to allow this engine to efficiently switch security contexts in the Java runtime as it impersonates multiple users. This requires modifications to Kerberos libraries and new GSF interfaces for establishing and cleaning up new user-specific security contexts.

## **8 Customers**

The GSF is currently being used by a number of projects, some of which are discussed below.

### **8.1 *Distributed Resource Management***

The Distributed Resource Management project (DRM) [15] is developing and deploying software products to simplify how users interact with high performance computing resources. DRM software aids in the discovery, reservation, allocation, monitoring, and control of geographically distributed computational resources throughout the DOE laboratories and production plants. These resources consist of heterogeneous compute nodes, communication, storage, visualization, data, and software. A portion of the system is CORBA-based and has been secured using the GSF. DOE recently accredited this application for operation on the classified networks linking the three DOE research laboratories.

### **8.2 *SAMPL***

The SAMPL (Simplified Analytical Model of Penetration with Lateral Loading) application provides Web-based, desktop access to a variety of penetration analysis and simulation codes. The system uses a signed applet that communicates via sockets with a compute server. The compute server, in turn, runs the necessary code or dispatches the request to another server as necessary. The system has been secured using the GSF socket extension.

### **8.3 *CoMPASS***

The CoMPASS (Confederation of Models to Perform Assessments of Stockpile Stewardship) project is creating a distributed, integrated model of the DOE Complex. This model will be used to help predict the consequences of decisions about issues ranging from dismantlement requirements to refurbishment schedules to capital investments, helping decision-makers evaluate whether the DOE Complex can meet changing demands with anticipated resources over the next 10 to 30 years. The software has been implemented with Java sockets and is in the process of being secured with the GSF socket extension. When fully deployed, this application will be widely distributed, accessing resources at many of the DOE laboratories and production plants that are located throughout the country.

## **9 Future Work**

Future areas of focus for the GSF include a mixture of research and customer support. We plan to enhance the GSF as needed in order to accommodate our customers and to build on additional platforms as necessary. We also plan on investigating the possibility of using the GSF to secure peer-to-peer systems, which represent a shift in the development of distributed systems.

## 10 Conclusion

The Generalized Security Framework provides a set of libraries and classes that can be used to easily secure distributed applications written in C++ or Java, independent of the underlying middleware or protocol. Extensions to the GSF are also available that make securing applications using Java sockets or Java RMI trivial. This allows developers of distributed applications to focus their energy on the task at hand without worrying about security. The GSF currently works with either DCE or Kerberos as the underlying security mechanism, but has been designed so additional security mechanisms can be added easily. The GSF provides authentication, authorization, data protection, auditing, and delegation services and meets the stringent requirements for operation on classified networks within the DOE complex.

## 11 References

- [1] J. I. Schiller, "Secure Distributed Computing," *Scientific American*, November, 1994.
- [2] Kohl, J., and C. Neuman, *The Kerberos Network Authentication Service (V5)*, IETF RFC1510, September 1993. <http://www.ietf.org/rfc/rfc1510.txt>
- [3] *DCE*, <http://www.opengroup.org/dce/index.html>, The Open Group.
- [4] *Kerberos: The Network Authentication Protocol*, <http://web.mit.edu/kerberos/www/>, Massachusetts Institute of Technology.
- [5] J. Linn, *Generic Security Service Application Program Interface*, IETF RFC1508, <http://www.ietf.org/rfc/rfc1508.txt>, September, 1993.
- [6] J. Linn, *Generic Security Service Application Program Interface, Version 2, Update 1*, IETF RFC2743, <http://www.ietf.org/rfc/rfc2743.txt>, January, 2000.
- [7] "Step-by-Step Guide to Kerberos 5 (krb5 1.0) Interoperability", Microsoft Corporation, December 2000, <http://www.microsoft.com/windows2000/library/planning/security/kerbsteps.asp>
- [8] "The Globus Project", <http://www.globus.org>
- [9] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, V. Welch., *A National-Scale Authentication Infrastructure*, IEEE Computer, 33(12):60-66, 2000.
- [10] S. Liang, *The Java Native Interface Programmer's Guide and Specification*. Addison-Wesley, 1999.
- [11] *Java Native Interface*, <http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html>, Sun Microsystems, Inc
- [12] *Java Remote Method Invocation (RMI)*, <http://java.sun.com/products/jdk/rmi/>, Sun Microsystems, Inc.
- [13] <http://www.corba.org>, Object Management Group.
- [14] *Creating a Custom RMI Socket Factory*, <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/rmisocketfactory.doc.html>, Sun Microsystems, Inc.
- [15] J. I. Beiriger, H. P. Bivens, S. L. Humphreys, W. R. Johnson, and R. E. Rhea, 2000, "Constructing the ASCI Grid," *Ninth IEEE International Symposium on High Performance Distributed Computing*, August, 2000.

## Distribution:

1 MS 0310 A. L. Hale, 9220  
1 0660 J. A. Larson, 9519  
1 0661 M. K. Bencoe, 9512  
1 0661 R. N. Harris, 9512  
1 0806 C. D. Brown, 9332  
1 0806 D. A. Hansknecht, 9332  
1 0806 P. C. Jones, 9332  
1 0806 Glenn Machin, 9332  
5 0807 Rich Detry, 8920  
1 0820 P. F. Chavez, 9232  
1 1137 J. L. Mitchiner, 6534  
1 1137 K. C. Bauer, 6534  
3 1137 Steve Kleban, 6534  
1 1137 W. A. Stubblefield, 6534  
1 1137 K. L. Heibert-Dodd, 6535  
1 1137 H. P. Bivens, 6535  
1 1137 S. L. Humphries, 6535  
3 1137 P. C. Moore, 6535  
1 1138 B. N. Malm, 6531  
1 1138 R. L. Vandewart, 6535  
1 1140 S. G. Varnado, 6500  
1 9003 K. E. Washington, 8900  
1 9011 Barry Hess, 8910  
1 9012 Paul Nielan, 8920  
1 9012 E. J. Friedman-Hill, 8920  
1 9012 Carmen Pancerella, 8920  
1 9012 Bob Whiteside, 8920  
1 9012 Christine Yang, 8920  
1 9012 S. C. Gray, 8930  
1 9012 K. R. Hughes, 8990  
1 9019 B. A. Maxwell, 8945  
1 9019 Ray Trechter, 8945  
1 9217 M. L. Koszykowski, 8950  
1 9217 P. T. Boggs, 8950  
3 9018 Central Technical Files, 8945-1  
1 0899 Technical Library, 9616  
1 9021 Classification Office, 8511/Technical Library, MS 0899, 9616  
1 9021 Classification Office, 8511 For DOE/OSTI